# NEURON® CHIP-based Installation of LONWORKS™ Networks

April 1992 **LONWORKS Engineering Bulletin**

## Introduction

Echelon's LONTALK™ protocol provides a flexible means of implementing control networks based on the NEURON CHIP intelligent communications processor. This flexibility means that the application designer can make appropriate tradeoffs among simplicity, functionality and cost in the design of the nodes forming the network. This is especially important in designing the installation scenario for the control network. A pre-installed network offers maximum simplicity and no flexibility — it functions as manufactured, and normally is not modified at a later time. At the other extreme, a network that is installed with a fully-featured network management tool can offer freedom to use all the features of the LONTALK protocol, at the cost of user interface hardware and a separate processor platform to execute the network management functions.

Network management tools based on the NEURON CHIP are typically used in cost-sensitive applications, where general purpose user interfaces and databases are inappropriate. There is no single best way of designing such a tool; the designer must make the tradeoff between cost, ease-of-use and functionality. Simplifying assumptions about the type of node to be installed, the network topology, and the flexibility of interconnection of network variables can often reduce the requirements on the database and the user interface. The challenge is to devise a user interface that is intuitive and avoids the possibility of non-obvious user error. Overloading excess functionality on primitive user interface hardware may allow the designer to claim that the device is fully featured, at the risk of making these features inaccessible to the average untrained user. An example of this is to be found in the user interfaces for programming available on some low cost VCRs.

It is not straightforward to specify simple design guidelines for network management tools in terms of numbers of nodes or numbers of channels. It is safe to say that a network requiring the use of configured routers probably cannot be installed by a NEURON CHIP-based network management tool because of the complex topology analysis required to determine the protocol timing parameters, the correct downloading order and the routing tables. However, networks with large numbers of nodes and simple logical topologies may be installed with a NEURON CHIP-based network management tool.

Table 1 shows a simplified classification of the different approaches to LONWORKS network management. It shows the tradeoffs among flexibility, cost and ease-of-use. Note that a network management tool does not need to be present for the network to function correctly — its task is to install and configure the nodes initially, and it need be present only when nodes need to be replaced or the connectivity of the network modified in any way.

| Installation Scenario | Network Mgmt Platform | Additional Cost per Node | Cost for Net Mgmt Tool | Installation Flexibility |
|---|---|---|---|---|
| Pre-installed | None | None | None | None |
| Self-installed | NEURON CHIP | <$10 | None | Minimal |
| NEURON CHIP-based | NEURON CHIP | <$10 | $1-$50 | Moderate |
| µP-based | Custom µP HW | <$10 | $100-$500 | Moderate |
| DOS API-based | PC compatible | <$1 | $500-$3,000 | Maximum |
| LONBUILDER™-based | PC compatible | <$1 | $15,000 | Full custom |

**Table 1.** Network Installation Scenarios

This engineering bulletin addresses the issues involved in developing network management tools based on the NEURON CHIP platform. The essential tradeoff is whether to include user interface hardware to support network management with each application node (for self-installation), or to create a special node with an easier-to-use more functional user interface to act as the network management tool (for NEURON CHIP-based installation). In the first case, there is an extra cost for each node for the user interface hardware, even if it is as simple as push-buttons or DIP switches. Also, each self-installing node requires extra program code to handle the logic of the self-installation process. For the second case, there is the requirement that a network management tool be present when the network is installed. With a network management tool, there is an initial cost associated with the user interface hardware and software for that node, but each additional application node requires very little or no extra hardware to support its installation.

## Network Management, Network Control and Network Monitoring

It is important to distinguish between network management, network control and network monitoring. Network management is simply the capability to install and configure the nodes in the network. A network management tool does not participate in the exchange of application messages and network variable messages, and so does not need to be present for the network to operate.

Using LONWORKS technology, any node can send and receive messages and network variables from any other node on the network. When one of the nodes in the network is the source or destination of most of the application messages, and the other nodes communicate only with this central node, then the network architecture is said to be centrally controlled. A network controller is this central LONWORKS node that coordinates the sense and control processing of the system. LONWORKS networks are frequently designed so that a network controller is not required, but instead use peer-to-peer communications This has the advantage that the system is not vulnerable to failures of any single node.

A network monitor is a node that receives application messages or network variable updates from many of the other nodes on the network. Any node in the network may be the destination of LONTALK packets from other nodes, and so may act as a network monitor. Network monitors can receive only network traffic that is addressed to them.

Central controllers and monitors frequently need to be connected to many other nodes. A node whose application program runs on the NEURON CHIP may define up to 62 network variables and up to 15 message tags. Up to 4,096 network variables may be defined on a node whose application program runs on a host processor, using the NEURON CHIP as a communications processor. In this case, the NEURON CHIP runs a special application program called the LONWORKS Microprocessor Interface Program (MIP). Figure 1 shows block diagrams of these two types of application nodes.

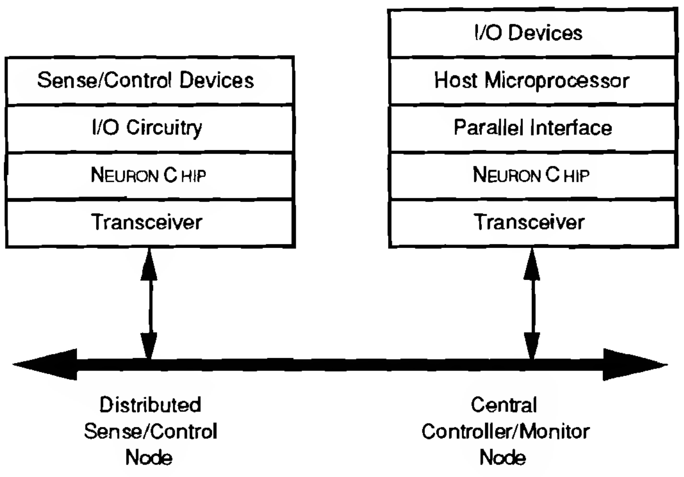


**Figure 1.** Application node block diagrams.

The design of network controllers and monitors is not covered in this engineering bulletin. However, the functions of network management (installation and configuration) and network control and monitoring (central application services) are frequently combined in the same node because of their common need for user interface hardware, and the database that contains the addresses of nodes and variables on the network.

## Installation Based on NEURON CHIPs

It is convenient to distinguish between two sub-cases of NEURON CHIP-based installation:

- Self-installed nodes update only their own network configurations based on user interface inputs — they do not interact with other nodes on the network during the installation process. This installation scenario simplifies the process by limiting the flexibility of the resulting network.

- A second scenario is when the network includes one or more NEURON CHIP-based nodes that have the ability to install other nodes on the network as well as themselves. These nodes are effectively *ad hoc* network management tools that implement some subset of the full network management functionality. A network management tool requires application code and database memory. If the tool is implemented on a NEURON CHIP, this code and data must fit into the 42K byte user address space of the NEURON 3150™ CHIP. If the tool is implemented on a host processor that uses the NEURON CHIP as a communications processor, then the resources of the host processor can be used for this code and data.

This class of installation tools is typically implemented with *ad hoc* user interface devices rather than computer-type user interface devices such as CRT displays and alphanumeric keyboards. These include devices such as push-buttons, DIP or rotary switches, LED indicators, annunciators, LCD or plasma numeric, alpha-numeric or graphic display panels, bar-code readers and special function or numeric-only keypads. Existing user interface devices already present on the node may be overloaded with an installation function, for example a rotary quadrature knob may be used to enter numbers if there is a numeric display for user feedback. A keypad may be used to enter installation commands in a special mode, or a bar-code reader may be used to enter location identification information. Detailed application examples of simple NEURON CHIP-based installation follow later.

Other installation scenarios are summarized here but will not be discussed in detail. For a general overview of installation issues, see the LONWORKS Engineering Bulletin *LONWORKS Installation Overview*, 005-0006-01. Figure 2 shows sample block diagrams of network management tools based on the NEURON CHIP, suitable microprocessor hardware, and the LONMANAGER™ API for DOS.

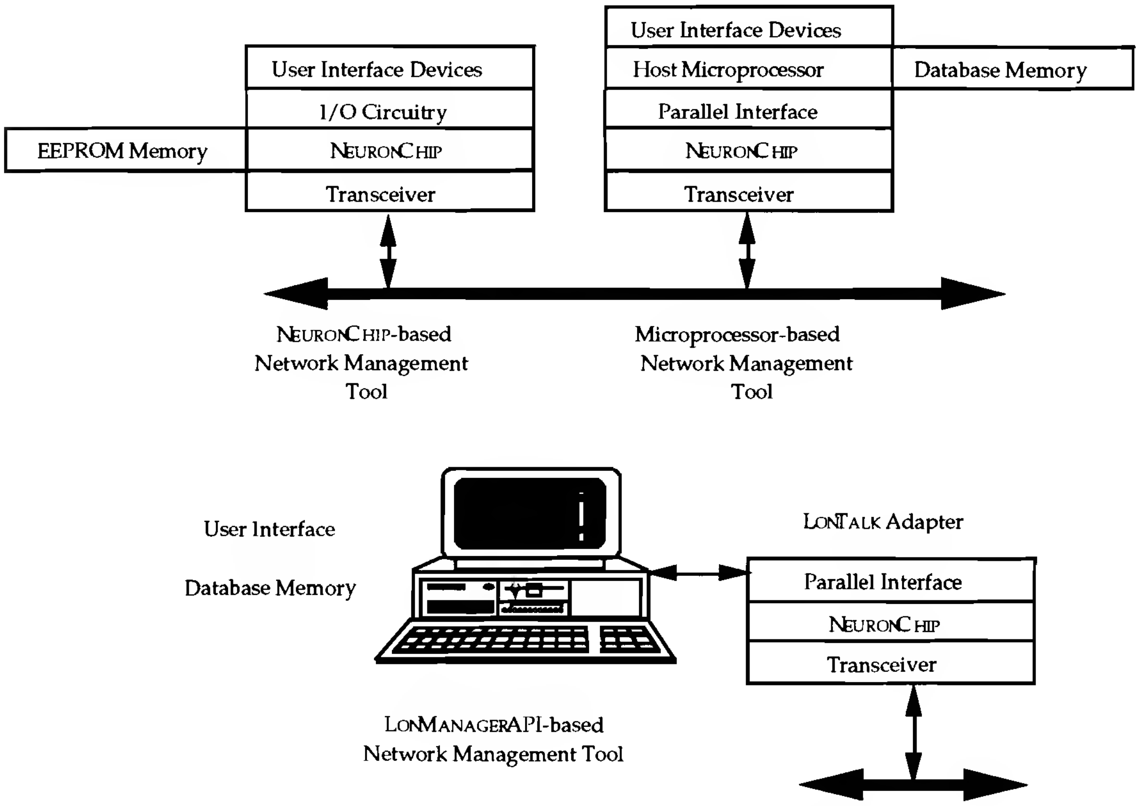


**Figure 2.** Network management tool block diagrams

## Pre-Installed Networks

Certain networks can be pre-installed at manufacture time, and therefore require no installation tools. For example, a closed network used to implement communications within an office machine can be configured when the machine is assembled. All models of a particular machine can use identical addressing and binding information, since they will not have to co-exist on the same network with other copies of the same machine. Also, pre-packaged collections of nodes may be pre-installed at manufacture time, and physically installed as a single system without any need for configuration. Such a pre-installed subsystem can be designed so that it may be incorporated into the database of a network management tool — this allows even simple networks to be expanded at a later date without having to replace the existing hardware.

## LONBUILDER-based Installation

Other networks may be installed using the LONBUILDER Developer's Workbench as an installation tool — this may be appropriate for high-end engineered systems where some degree of field customization of the application programs is desired. A professional system installer could use the LONBUILDER Developer's Workbench on-site during the commissioning of the network — it does not need to be present for the network to function. The LONBUILDER Developer's Workbench also provides a protocol analyzer that is convenient for diagnosing network traffic and throughput problems on multi-media networks. The user interface of the LONBUILDER Developer's Workbench provides access to all the tunable parameters of the LONTALK protocol, together with a LON® database that keeps track of the nodes on the network and the network topology.

## LONMANAGER API-based Installation

Fully-featured network management tools are based on general-purpose micro-processors or computers, not on NEURON CHIPS. For details, see the LONMANAGER API product documentation. This class of installation tool is typically implemented with graphics displays, pointing devices, full alphanumeric keyboards, and non-volatile database storage — devices appropriate to a PC-class computer. The installation scenario for an API-based tool could even include downloading of installation-specific object code modules into the application nodes. The nodes themselves need minimal extra hardware, typically just the ability to ground the service pin of the NEURON CHIP at installation time, or to respond to the *wink* network management message.

The LONMANAGER API provides a superset of the functions provided by the network manager integrated into the LONBUILDER Developer's Workbench. The LONMANAGER API provides these functions in object code form that can be integrated into customized network management tools. These functions include the database that keeps track of the state of the network, network topology analysis to calculate and download protocol parameters, application node configuration, and router configuration for complex multi-media networks. The LONMANAGER API also allows the network management tool to act as an application node or network controller — it can generate and receive network variable updates and other application messages. These functions are logically separate but are often combined in one device.

## Installation Tasks

As an example of the tasks involved in installing a network, the process of network variable binding will be discussed here. Network variables are programmer-defined interface objects for the nodes in the network. Binding is the process of creating logical connections between network variables on different nodes. For example, figure 3 shows an output network variable on Node 1 associated with input network variables on Node 2 and Node 3. When the application program running on Node 1 updates the output network variable, a message is sent across the network to Node 2 and Node 3 to cause the values of the input network variables to be updated. For a detailed discussion of how the programmer specifies this, see Chapter 3 of the *NEURON C Programmer's Guide*.

In order to make use of the full flexibility of the LONTALK protocol, there are other attributes of the communications between nodes that can be configured during installation, for example, the use of subnets and routers. These topics are covered in *LONTALK Protocol* Engineering Bulletin, 0005-0017-01.

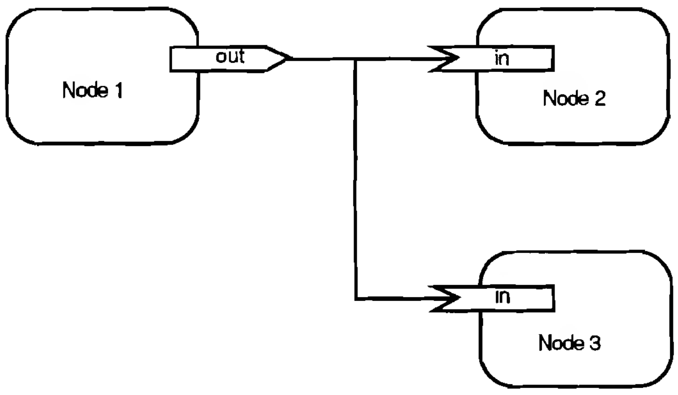


**Figure 3.** Example of network variable binding

The network variable objects on the nodes are defined when the application program is written. However, the programmer does not specify the connections between the network variables, or how other nodes are to be addressed. This is done when the nodes are installed in a network. This separation of application function and addressing allows nodes to be manufactured in quantity, and then customized to the particular environment in which they are to be installed.

Each node contains tables in its EEPROM memory that define the node's addresses on the network and the connection information necessary for the node to be able to propagate its output network variables to other nodes. The contents of these tables are defined during installation — how this happens is determined by the installation scenario. When nodes are pre-installed using the LONBUILDER Developer's Workbench, or when they are installed with a network management tool based on the LONMANAGER API, the complexity of this addressing and binding mechanism is transparent to the system designer — binding happens automatically. However, for nodes that are self-installing or use NEURON CHIP-based installation scenarios, this complexity has to be understood and taken into account when designing the application. Before presenting examples of hardware and software that implement NEURON CHIP-based installation, it is important to understand the mechanism used by the LONTALK protocol to deliver network variable updates.

## Network Variable Update Packet

The LONTALK protocol implements network variable updates by sending a packet from the source node to the destination nodes when a network variable value is propagated across the network. Figure 4 shows a simplified view of the information in this kind of packet. For a more detailed description, see the *LONTALK Protocol* Engineering Bulletin.

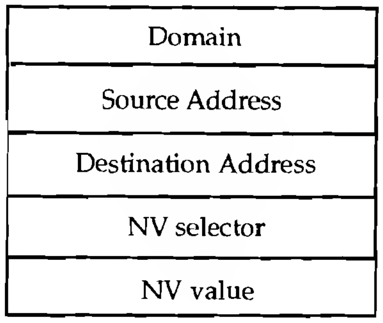


**Figure 4.** Network Variable Update Packet

The fields in this packet are created by the source node, and interpreted by all the other nodes on the network that receive the packet at the data link layer (OSI layer 2).

## Domain

Packets on the network are sent in a particular domain. Each node on the network belongs to one or two domains, and can only send and receive application packets such as network variable updates on a domain that it belongs to. This is so that multiple sub-systems can co-exist on a single network — as long as they use different domain identifiers, the sub-systems cannot interfere with each other. A domain consists of one or more subnets, and nodes that belong to the domain are assigned to a subnet in that domain.

## Source Address

This field contains the identifier (subnet number within the domain, and node number within the subnet) of the node that generated this packet — it is used by the destination nodes to send any acknowledgements that are required. The subnet number is also used by routers for routing the packet.

## Destination Address

This field contains the address of the nodes to which this packet is addressed — there are three formats for this field depending on whether a single destination node or multiple destination nodes are being addressed. In the first case, called unicast or subnet-node addressing, the destination address is the subnet and node number of the destination node. In the second case, called multicast or group addressing, the destination address is a group number. Nodes may be defined to be members of different groups within each domain so that multiple input network variables can be updated with a single update message. In the third case, called broadcast addressing, the destination address is a whole subnet or the whole domain, and only unacknowledged or unacknowledged repeated service is available.

### Network Variable Selector

This is a 14-bit number used to uniquely identify the network variable when it arrives at the destination node. The only requirement is that different network variables on the destination node have different selectors, so that when network variable update messages arrive at the destination node, they are vectored to the correct network variable. One way to ensure this is to make the selectors unique across the whole network, but this is not required. Selector values 0x3000 - 0x3FFF are reserved for unbound network variables, with the selector value equal to 0x3FFF minus the network variable index. Selector values 0 - 0x2FFF are available for bound network variables. A binder program in the network management tool allocates network variable selectors to achieve the desired connectivity.

### Network Variable Value

This is the actual data that is being propagated in the network variable. It may be from one to 31 bytes long. The destination node copies this value into the input network variable when the update packet is received successfully.

## Source Node Data Structures

The source node is responsible for filling in the fields in the network variable packet described in figure 4. It does this by referring to internal configuration data structures that were established when it was installed. A pre-installed node, or a node installed with a network management tool will have these fields filled in by the LONBUILDER Developer's Workbench, or by the network management tool. A node that is self-installing will need to modify these data structures from within its NEURON C application program in order to achieve the desired effect. Figure 5 shows a simplified view of the data structures in the source node that are involved in creating network variable update packets. For more detail on these data structures, see Appendix A of *NEURON CHIP Advance Information,* 005-0018-01.

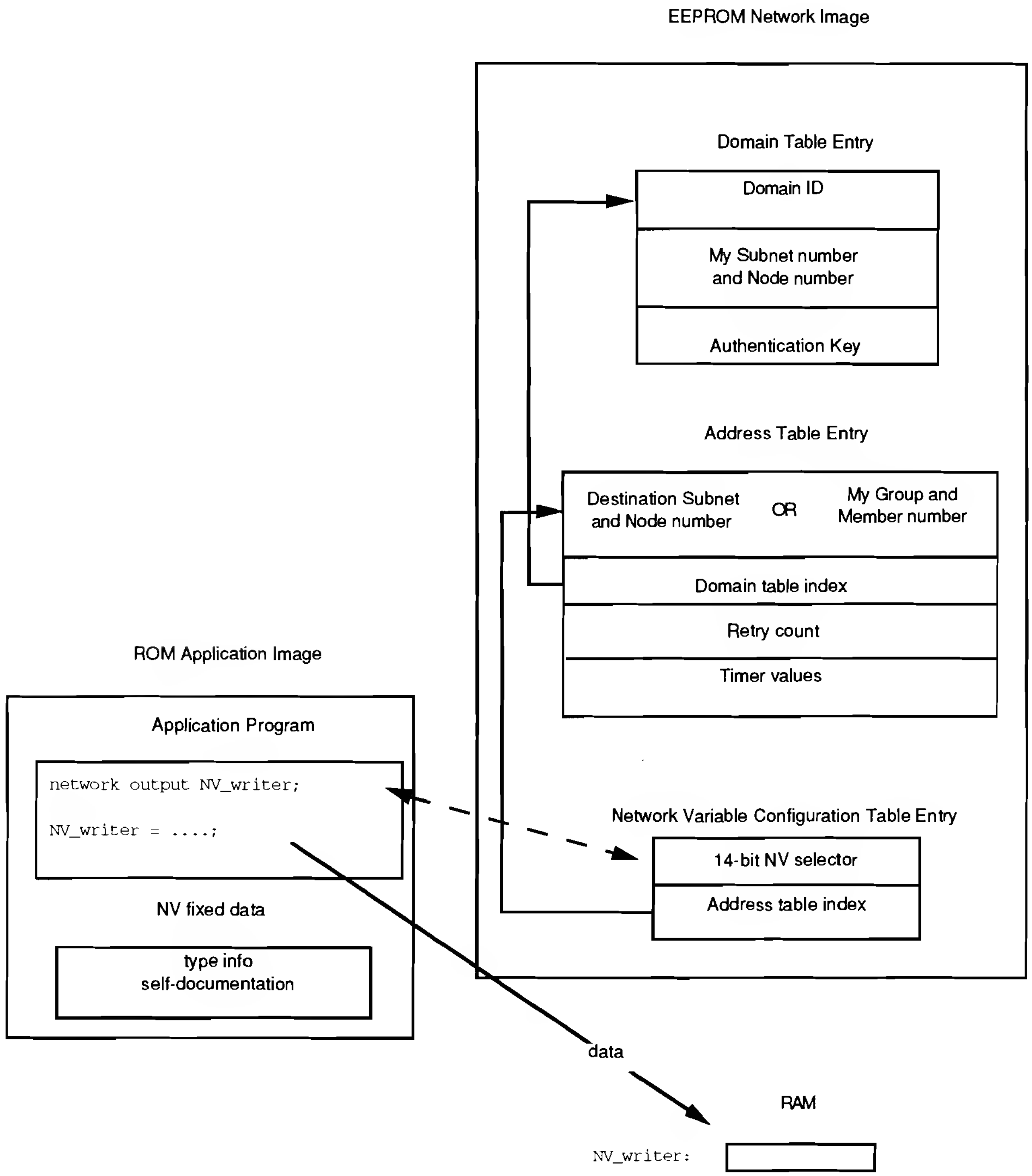


**Figure 5.** Source node data structures

## Network Variable Configuration Table

Every output network variable is assigned a location in RAM to hold its value and an entry in the EEPROM-resident network variable configuration table. This entry specifies the 14-bit network variable selector to be used when the value is propagated across the network, and also a reference to an entry in the address table, which specifies the destination address for the update. The table also specifies the service class (acknowledged, unacknowledged, or unacknowledged repeated), and whether authentication or priority should be used. Network variables may also optionally have self-identification and self-documentation information stored with the application image. This information can be retrieved over the network by an intelligent network management tool. See *NEURON C Programmer's Guide Supplement* page 19 for information on specifying self-identification and self-documentation for network variables.

### Address Table Entry

An address table entry is used by the source node to specify a destination address for a network variable update. It contains either the subnet-node, group or broadcast address of the destination nodes for the update. Unique 48-bit NEURON ID addresses are never used for network variable updates — this reduces the size of each message, and also permits faulty nodes to be replaced without having to update all other nodes that communicate with the failed node. The address table entry also contains transport layer messaging parameters such as the retry count and various timer values used to control the protocol service. Each address table entry also refers to a domain table entry specifying the domain on which the update is to be delivered. On a single node, there may be up to 15 address table entries that are used for implicitly addressed messages and network variable updates.

### Domain Table Entry

A domain table entry contains a domain identifier of length 0, 1, 3 or 6 bytes, the subnet and node numbers of the node relative to the specified domain, and the authentication key for the specified domain. Each node may be a member of one or two domains, and the domain identifier is inserted into each packet propagated onto that domain. If the network variable update is authenticated, the authentication key for that domain is used when the source node constructs replies to the challenges received from the destination nodes.

## Destination Node Data Structures

The destination node is responsible for interpreting the fields in the network variable packet described in figure 4. It does this by referring to the internal configuration data structures that were established when it was installed. Figure 6 shows a simplified view of the data structures in the destination node that are involved in processing network variable update packets. These are the same data structures that are used when the node is acting as a source node and any node can both send and receive network variable updates.

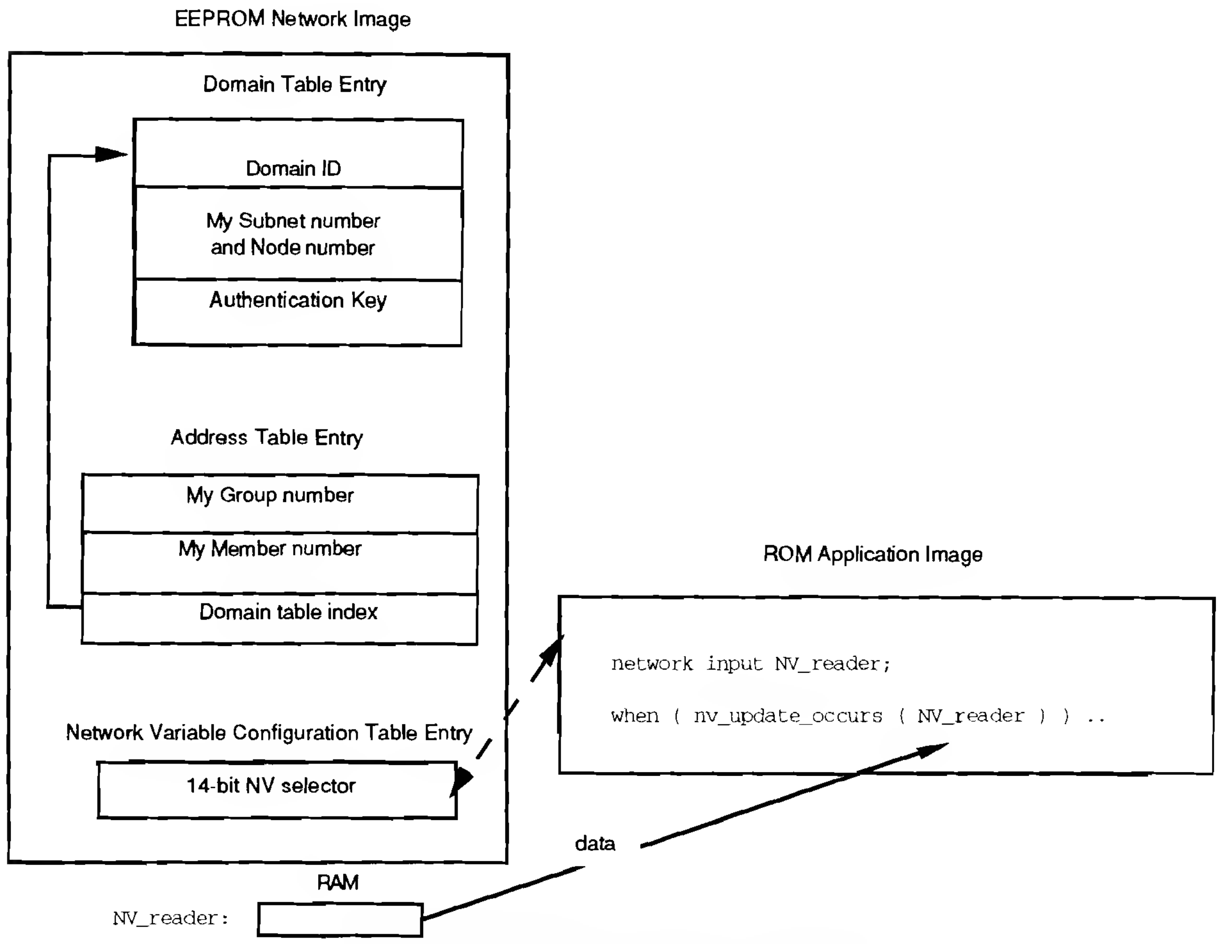


**Figure 6.** Destination Node Data Structures

## Domain Table Entry

For each packet that arrives at the destination node, the domain of the packet is checked for a match with one of the entries in the domain table. If the packet is in one of the domains that the node is in, then the next action depends on the address format of the packet. If it is addressed using subnet-node addressing, then the destination subnet and node numbers are checked. If it is addressed using group addressing, the address table is checked to see if the node is a member of the specified group. If the packet passes one of the address checks, or if it is a broadcast packet, then the message is further processed. If the network variable update is to be authenticated, the authentication key for the domain is used to construct the challenge to the source node. If the authentication succeeds or if no authentication was requested, the network variable is updated.

## Address Table Entry

For group addressed network variable updates, the address table entries indicate if the destination node is a member of the group. The member number in the corresponding address table entry is used to construct an acknowledgement, if one is required. Acknowledgements are delivered using subnet-node addressing back to the source node. This way the source node can keep track of which of the destination nodes have acknowledged the update and can issue reminder network variable update packets if necessary.

### Network Variable Configuration Table Entry

Once the packet has been accepted as being validly addressed to the destination node, the network variable selector is compared against all the selectors in the network variable configuration table. If a match is found, then the local value of the corresponding input network variable is updated in RAM, and the appropriate `nv_update_occurs` event posted to the application program. If a network variable update arrives at a node that has no matching selector in its network variable configuration table, the update message is ignored, although the proper acknowledgements are generated. NEURON C applications can access the source address of the network variable update packet using the `nv_in_addr` built-in variable. Applications using the Microprocessor Interface Program (MIP) can also access the source address.

## Addressing, Binding and Configuration

At the simplest level, an installation scenario for network variable updates has to specifying four pieces of information: assigning a domain, assigning subnet and node identifiers for each node in the domain, assigning address table entries for each node, and assigning network variable selectors to implement the desired binding. Several possibilities exist for each of these in the context of a network management tool.

### Assigning a Domain Identifier

All nodes have to become members of this domain so that they can communicate with each other. Possibilities include the following:

- No domain identifier — useful on closed networks where no other subsystem uses the same medium.
- Domain identifier assigned by a user interface on the network management node. The installer is responsible for ensuring uniqueness. In the self-installation example, the user interface is one of the thumbwheels on each of the application nodes. With the LONBUILDER Developer's Workbench, the user specifies domain identifiers by typing the values into the *Domain Create* dialog box.
- Each subsystem can pick a domain identifier and hope that it is unique. This has obvious problems.
- The domain identifier can be chosen to be the unique 48-bit NEURON ID of one of the NEURON CHIPs in the network, for example, the NEURON CHIP in the network management tool. This guarantees uniqueness, but has the

disadvantage that all messages use a long six-byte domain identifier, which increases the overhead associated with each packet.

## Assigning Subnet and Node Identifiers

Possibilities include the following:

- Subnet and node identifiers are not assigned at all, as shown in the self-installation example. This is possible if unacknowledged or unacknowledged repeated service is used with multicast addresses exclusively. If the network uses configured or learning routers, then assigning subnet identifiers is always necessary.

- The network management tool can assign subnet and node identifiers to each node as it is installed and keep track of the identifiers already used. This information can be kept in a non-volatile memory, for example.

- The network management tool can pick random numbers as IDs, and then query the network to determine if these IDs have been used. This process is called *hailing* and has the disadvantage that it may not be a reliable way of preventing duplicate IDs being issued, since there is a possibility that some of the nodes may be powered down and thus unable to respond to the hail.

Once the network management tool has determined an appropriate subnet and node identifier for a newly installed node, it loads that information into the target node's domain table using the *Update Domain* network management message.

## Acquiring Node Addresses

Normally, network management messages are sent to the target nodes using their subnet and node identifiers as destination addresses in the domain of the application. However, the first network management message that makes the target node join the domain cannot be delivered this way since the node is not yet a member of the domain. This first message is delivered using unique 48-bit NEURON ID destination addressing mode. The network management tool has several ways of acquiring the unique 48-bit NEURON ID from a newly installed node.

The NEURON CHIP in each node has a service pin which, when grounded, causes the node to transmit a message containing its unique 48-bit NEURON ID as a domain-wide broadcast on the domain whose ID is zero bytes in length. The *Service Pin* message also contains the Program ID of the target node, which is an eight-byte identifier of the application program running on the target node.

A second way to acquire a node address is by issuing the *Query ID* network management message to all unconfigured nodes as a broadcast message. All unconfigured (domain-less) nodes will respond with a message containing their

48-bit NEURON ID and Program ID, and the network management tool can handle the first of these messages. For example, each node may be temporarily connected to the network management tool, which then issues the *Query ID* network management message. The 48-bit NEURON ID in the response is then associated with the node being installed.

Alternatively, if several nodes are installed together, there will be several responses to the *Query ID* message. The network management tool can then issue a *Wink* network management message to the node that generated the first response. In this case the application program on each node has a task which is activated by the receipt of the *Wink* message. This task can then cause the node to respond in a sensible fashion by activating some output device connected to the node's I/O pins, for example, with a visible or audible indication to the human installer. This, then, uniquely associates that physical node with its 48-bit NEURON ID known to the network management tool.

The third way to acquire the physical address of a node is to enter it into the network management tool manually. For example, a node may have a machine-readable bar code with its NEURON ID printed on the outside, or the 6-digit hexadecimal ID may be entered from a keypad into the network management tool.

## Assigning Destination Addresses

A subnet-node (unicast) address is the subnet and node identifier of the target node. But if group (multicast) addressing is desired, the network management tool needs to assign group identifiers in the domain and have target nodes join the different groups. It needs to ensure that the group identifiers are unique and to assign member numbers to each node that belongs to the group. Member numbers are required if acknowledgements are to be generated for network variable updates sent to members of the group.

Again, the network management tool can assign group identifiers and member numbers by keeping track of the numbers already used in a non-volatile database. Hailing is also a possibility for allocating group identifiers, subject to the same limitation discussed previously. When destination addresses are assigned, the network management tool loads that information into the target nodes' address tables using the *Update Address* or the *Update Group Address* network management message.

## Assigning Network Variable Selectors

The network management tool has to assign network variable selectors to implement the desired connections. How this is done depends on the degree of flexibility required. The simplest case is when there is only one possible way to connect two nodes together. In this case, the LONBUILDER Developer's Workbench can be used to assign the selectors, as in the following example. If more flexibility is desired, the network management tool acquires the desired connections from some user interface and allocates the selectors appropriately. The network management tool keeps track of the assigned selectors so that they are unique within any one node. When network variable selectors are assigned, the network management tool loads that information into the target nodes' network variable configuration tables using the *Update NV Config* network management message.

## The LON Database

In general, the network management tool needs to keep track of the network as it is being installed and modified. The information needed includes the domain ID for this application, which subnet and node IDs have been used, which group IDs have been used, which member numbers have been assigned for each group, and which NV selectors have been used.

Simple network management tools can choose to keep this data in a non-volatile memory. Alternatively, the human installer may be made responsible for ensuring the uniqueness of the various IDs with the risk that an error may be made leading to undesirable behavior of the network. The network management tool may also be programmed to extract this information from the network each time it is needed. This has the disadvantage of possibly being time-consuming and unreliable if part of the network is not functioning.

A network management tool based on the LONMANAGER API uses a disk-resident LON database to keep track of this data, as well as other data on the application node hardware, application programs, node specifications, channel parameters and topology, and router configurations. This considerably simplifies the task of installing a complex multi-media network with various different types of nodes. The size of the LON database in a NEURON CHIP-based network management tool depends on the degree of flexibility required in the installation scenario, the performance needed for searching the database, and the number of nodes, groups, channels and other objects.

## Binding

A binder is a program that assigns network variable selectors and address table entries on the nodes in the network to achieve the desired connectivity between network variables and message tags. A binder algorithm is a central part of a network management tool, and several implementation possibilities exist depending on the flexibility desired. For example, the binder that is incorporated in the LONBUILDER Developer's Workbench and in the LONMANAGER API allows overlapping connections, to the extent that an input network variable may participate in more than one connection. This binder assigns a network variable selector to the union of overlapping connections. If there is a single non-polled input network variable in the connection, it assigns a unicast address table entry on the nodes with output network variables in the same connection. If there is more than one input network variable in the connection, it assigns a multicast address table entry on all the nodes in the connection.

This is not the only possible way that a binder can operate. For example, if the limit of fifteen address table entries on any one node becomes a problem, then nodes may be grouped together in larger groups so that address table entries may be shared among several different network variables. This generates more network traffic because groups are larger than would otherwise be needed and so extra acknowledgements need to be sent. The examples presented in this engineering bulletin use very simple binding algorithms.

## Self Installation Example

In this scenario, each node is able to write to its own network configuration data structures, but does not exchange messages with other nodes to coordinate the binding process. The example deals with the basic steps of installation: assigning a domain, assigning subnet and node IDs, assigning destination addresses for the network variable updates, and assigning network variable selectors.

### Assigning a Domain identifier

To simplify the example, assume that the node has only one domain table entry (specified with the NEURON C compiler directive `#pragma one_domain`). The first task is to assign a domain identifier. There are two possibilities: if the network is closed (for example, a twisted-pair medium with no other subsystem on the same network), then the domain identifier may be zero-length, since there is no risk of a misdelivered message. No user interface needs to be provided for domain assignment. On the other hand, if the network is open or shared (for example, an RF or powerline medium, or a shared twisted-pair medium) then the domain identifier must be assigned by the installer. A simple user interface to input a single 4-bit number to a node could be a 16-position binary encoded thumbwheel switch (for example Digi-Key SW215-ND), or an array of four DIP switches, attached to four of the NEURON CHIP's I/O pins. This allows the installer to specify a single byte domain identifier in the range 0 to 15. The application program then has to read this array of switches, and set the domain identifier in the domain table.

### Assigning Subnet and Node Identifiers

For simplicity, all nodes belong to a single subnet which are assigned by the LONBUILDER Developer's Workbench when the nodes are developed. Multiple subnets are only important if configured or learning routers, or more than 127 nodes, are needed in the domain. Subnet and node IDs are also required for acknowledgements. Assigning the node identifier could be done with a second thumbwheel or DIP switch array on each node, but there still remains the issue of specifying connections between network variables. Instead, this example ignores the issue of assigning a unique node identifier, and uses only unacknowledged group addressing. In this case, group identifiers are used as destination addresses, and node identifiers are not used.

## Assigning Group Membership

In order for a node to belong to a group, it normally needs to know how many members there are in the group for the purposes of counting acknowledgements. This means that assigning a node to a group would require updating the address tables of all other members of the group, which contradicts the requirement that each node be self-installing. However, if unacknowledged or unacknowledged-repeated service is used exclusively, members of the group do not need to know how many other members there are. This example, then, requires that all network variables be specified with one of these two classes of service. This is done with the `bind_info` qualifier in the declaration of output network variables. The disadvantage of using unacknowledged or unacknowledged repeated service is that the source node has no way of knowing whether the destination nodes have successfully received the network variable update, or even if they exist on the network. On the other hand, using unacknowledged repeated service increases the likelihood that an update will get through even in the face of collisions on the network.

This example uses a second 16-position thumbwheel or 4-position DIP switch to associate each node with one particular group. Only one address table entry is created, and all network variable updates use this single entry. Overlapping connections are not permitted, so that the each node is a member of only one group. A switch matrix approach is used to scan the switches, allowing the NEURON CHIP to read the state of both sets of switches (one for domain, and one for group) using only six I/O pins. This leaves five pins for the application's I/O. The application program then self-configures the node to belong to the domain and group whose numbers (in the range 0-15) are specified by the appropriate set of switches. Figure 7 shows the schematic for this approach. Adding an inverter to this circuit can save one more I/O pin.

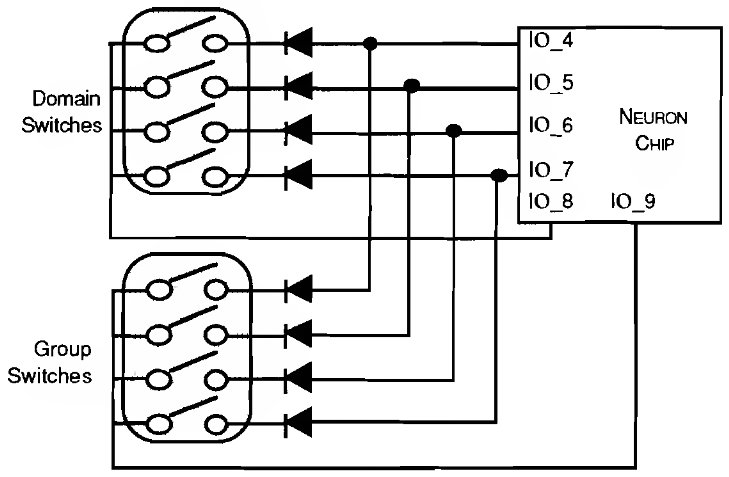


**Figure 7.** Sample user interface schematic to specify domain and group IDs for self-installing node

The self-installation software reads each of these switch arrays separately. If IO_8 is driven to logic 0, and IO_9 is driven to logic 1, the nibble port made up of pins IO_4 through IO_7 will read the domain switches. If, on the other hand, IO_9 is driven low, and IO_8 high, then the nibble port will read the group switches. Example code is shown in Listing 1.

```
#include <addrdefs.h>
#include <access.h>
#include <snvt_cfg.h>

#pragma enable_io_pullups
                  // enable on-chip pullup resistors on pins IO4 - IO7
#pragma one_domain
                  // only one domain
#pragma num_addr_table_entries 1
                  // only one address table entry

IO_4 input nibble IO_switches;           // declare switch array device
IO_8 output bit IO_domain_select;
IO_9 output bit IO_group_select;
network input config SNVT_config_src NV_self_config = CFG_LOCAL;

domain_struct my_domain;      // local copy of domain structure
address_struct my_addr_table; // local copy of address table structure

when( reset ) {
    unsigned switches;

    if( NV_self_config == CFG_LOCAL ) {

      io_out( IO_domain_select, 0 );      // select domain switches
      io_out( IO_group_select, 1 );
      my_domain = *access_domain( 0 );    // read domain table entry 0
      switches = io_in( IO_switches );    // read domain number from switches
      if( my_domain.id[ 0 ] != switches ) {     // see if there is any change
          my_domain.id[ 0 ] = switches;
          update_domain( &my_domain, 0 ); // update domain table entry 0
      }

      io_out( IO_domain_select, 1 );      // select group switches
      io_out( IO_group_select, 0 );
      my_addr_table = *access_address( 0 ); // read address table entry 0
      switches = io_in( IO_switches );    // read group number from switches
      if( my_addr_table.gp.group != switches ) {  // see if there is any change
          my_addr_table.gp.group = switches;
          update_address( &my_addr_table, 0 );  // update address table entry 0
      }
   }
}
```

Listing 1. Self-installing node; getting the domain and group identifiers

For a description of the functions `access_domain`, `access_address`, `update_domain` and `update_address`, see the *NEURON C Programmer's Guide.*

The configuration network variable `NV_self_config` is defined by the program with an initial value of `CFG_LOCAL` (0), and tested to see if this node should self-configure on power-up. If this node is ever installed by an external network management tool, then that tool should send a network variable update to set the variable `NV_self_config` to the value `CFG_EXTERNAL(1)`. This then prevents the self-configuration code from overriding the node's externally-specified configuration each time it is reset. The `config` qualifier causes the variable `NV_self_config` to be stored in EEPROM so that its value is preserved when the node loses power.

The code example avoids writing to the domain table and the address table if the state of the switches has not changed. Without this logic, the node would write these locations every time it was reset, potentially limiting the retention of the EEPROM memory, which has a specified lifetime of 10,000 write cycles. In this example, to save code space, it is assumed that the domain table entry and address table entry have been set up completely in EEPROM except for the domain number and the group number. These two numbers are overwritten by the self-installation procedure and all other values are left as is.

The other values can be set up using the LONBUILDER Developer's Workbench, and loaded as part of the network image when the node is manufactured. For example, to create the domain table entry, the LONBUILDER's *Domain Create* dialog box is used to create a domain with an ID size of one and a subnet in that domain. The *Appl Node Create* dialog box is then used to assign the node to that subnet. To create the address table entry, the *Connection Create* dialog box is used to create a connection from a network variable on the node to network variables on more than one other node using unacknowledged repeated service. This will ensure that an address table entry in group format is created with the appropriate transport layer messaging parameters.

## Assigning Network Variable Selectors

In this example, each node has only one address table entry corresponding to a group, and so all its network variable updates will use that group. If the nodes have multiple network variables, then they are differentiated by their selectors. For simplicity, assume that these can be assigned by the LONBUILDER Developer's Workbench when the node is developed, and so no user interface hardware or software is necessary to specify them. The developer builds a prototype network with the nodes interconnected as desired, and then the LONBUILDER binder allocates the appropriate selectors and creates the network image for each node's EEPROM. This assumption means that nodes belonging to the same group always have their network variables connected together in the same way. If this is insufficiently flexible, then some user interface must be provided to allow the user to specify the desired interconnect. Figure 8 shows an example of a network with two groups installed in a domain.

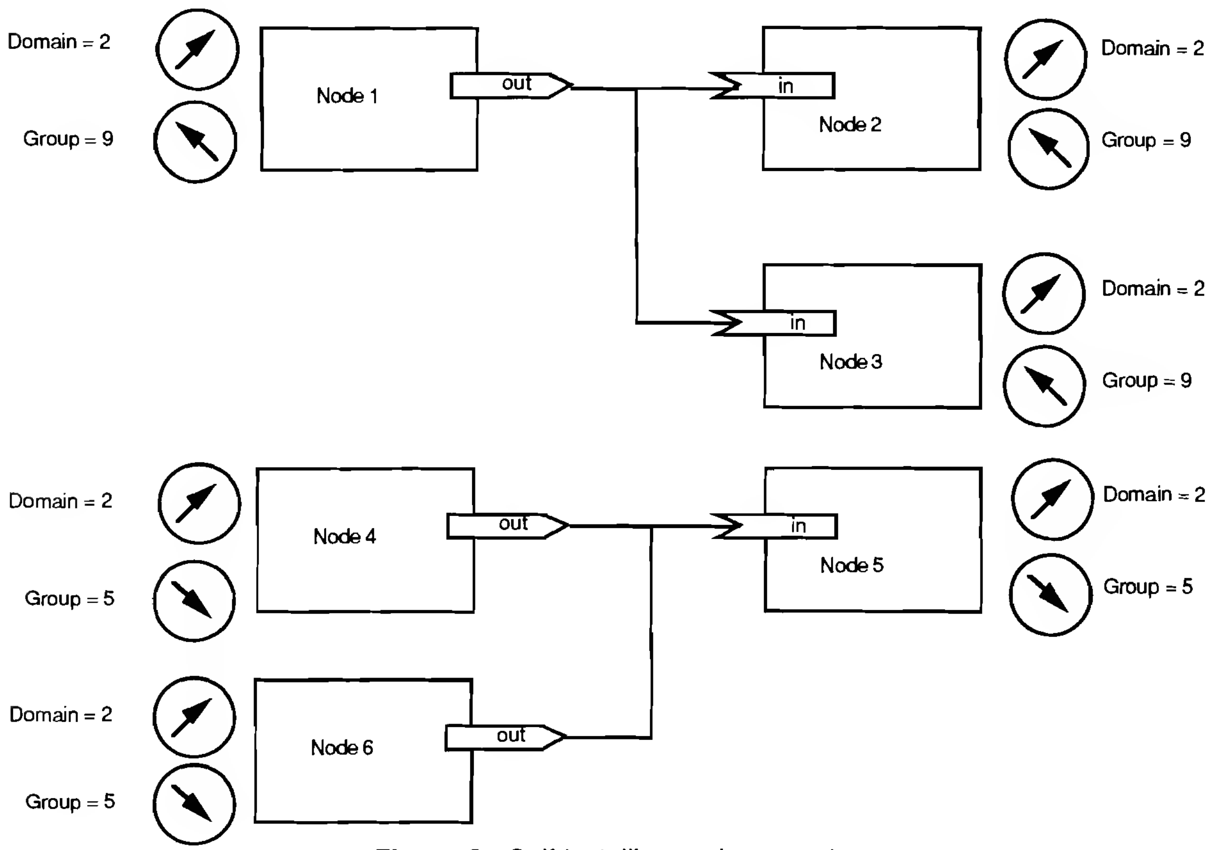
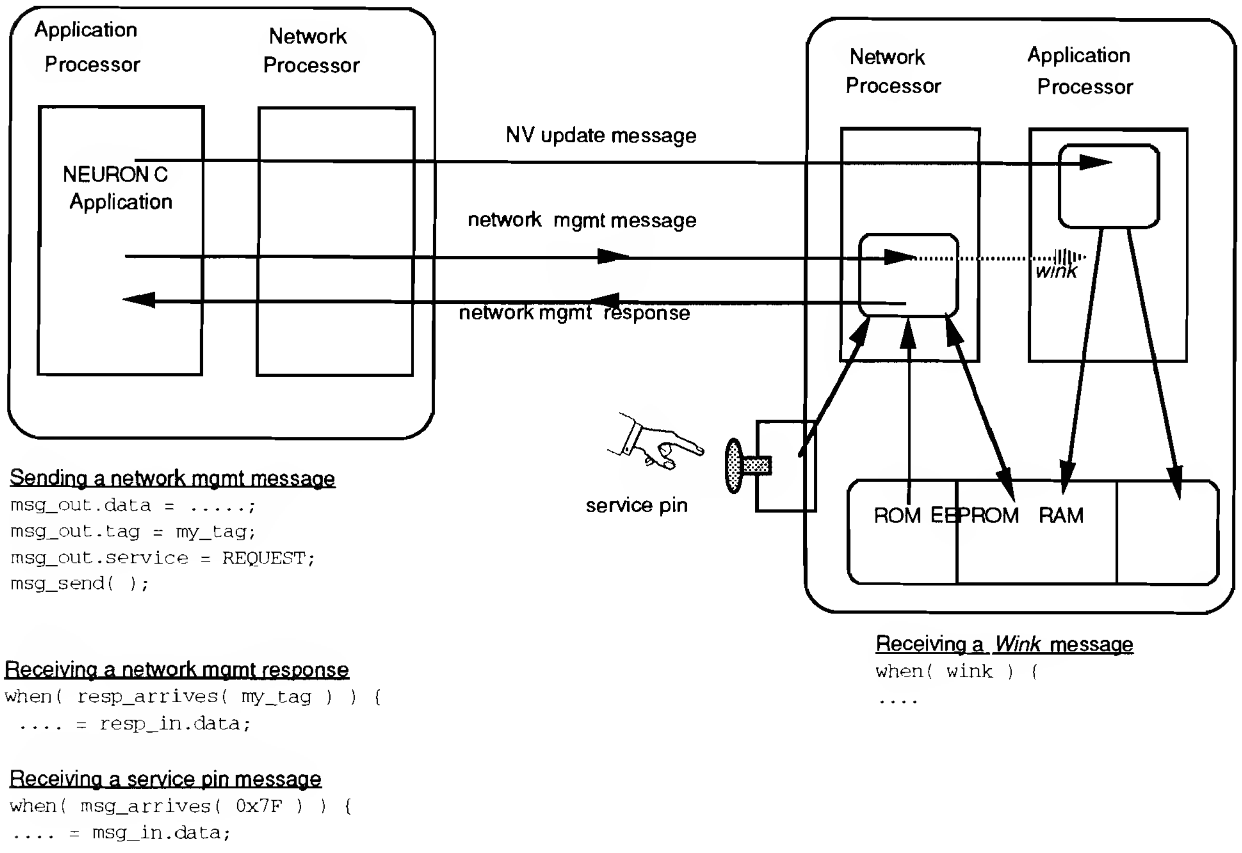


**Figure 8.** Self-installing node example

## Network Management Messages

The self-installation scenario presented above is simplified so that each node can install itself, and there are restrictions on the LONTALK protocol features that can be used in the application. In a more typical case, there is a network management tool that sends messages of a special type to the other nodes to effect a more general installation. Messages between nodes are either application messages or network management messages. Figure 9 shows how these two kinds of messages are propagated between nodes. Application messages, including network variable updates and polls, are generated by the application processor on the source node, and propagated over the network to the application processor on the destination nodes. Network management messages may be generated by the application processor on the source node, but most of them are handled by the network processor on the destination nodes. They include messages to query and update the domain, address and network variable configuration tables, as well as messages to reset the node, query its status, and change its state. Network management message to make the node go on-line or off-line, and to activate the *Wink* task are delivered to the application processor on the destination node where they post the corresponding events. The application processor on a node can receive the responses to any network management messages that are sent with request/response service, as well as receive an unsolicited service pin message from any node. In order to receive a service pin message, a node must belong to the zero-length domain.



Sending a network mgmt message

```
msg_out.data = .....;
msg_out.tag = my_tag;
msg_out.service = REQUEST;
msg_send( );
```

Receiving a network mgmt response

```
when( resp_arrives( my_tag ) ) {
 .... = resp_in.data;
```

Receiving a service pin message

```
when( msg_arrives( 0x7F ) ) {
 .... = msg_in.data;
```

Receiving a *Wink* message

```
when( wink ) {
....
```

Figure 9. Processing of application messages and network management messages

Application messages are exchanged between the application processors on two or more nodes on the network, for example, network variable updates and their acknowledgements used to share data between application programs. Network management messages are handled by the network processor on the destination node and are used for management functions such as installation and diagnostics. Network management messages include messages for reading and writing the domain table, the address table, the network variable configuration table, as well as the application program itself, if not already installed in the node. For details, see the *NEURON CHIP Advance Information*, Appendix B. In the NEURON CHIP-based installation scenario, the NEURON CHIP in the network management tool sends network management messages to configure the other nodes in the network.

## NEURON CHIP-based Installation Example

In this example, there is a single NEURON CHIP-based node that is responsible for installing the other nodes. It accepts service pin messages from other nodes in order to acquire their addresses and then assigns them subnet and node IDs. By pressing the service pins in a particular order, the installer indicates which nodes should be connected. This is a very minimal user interface, only a single push button attached to the service pin is required on each node. This example is intended to indicate the kind of functions that a network management tool has to perform, rather than to recommend any particular user interface scenario.

There are two kinds of nodes that are to be bound together. The sensor node type, whose program ID is SENSOR has a single output network variable. The actuator node type, whose program ID is ACTUATOR, has a single input network variable of the same type. A network can consist of any number of sensors and any number of actuators, and they can be associated in groups. Each sensor node and each actuator node may belong to one group at most, and the values of any of the sensor nodes' output network variables are propagated to all the actuator nodes' input network variables. In addition, there is a single NEURON CHIP-based network management tool — the NEURON C code for this node is presented here. Conceptually, this node may also be a sensor or an actuator, but that possibility is not considered here for simplicity reasons.

The network management node has a single push-button and a single LED labelled INSTALL. The user puts the network management tool into installation mode by pressing the push-button. This lights the LED. Then the user presses service pin push-buttons on any of the sensor or actuator nodes. The network management node collects the service pin messages, and configures the indicated nodes to belong to a single group. This also has the effect of removing those nodes from any previous group they may have belonged to. The installation process is terminated by pressing the push-button on the network management tool again, and the LED goes out. Other groups may be established by repeating the procedure.

The design of a possible implementation of this network management tool is now presented, by following the steps of domain, subnet, node, group and network variable selector assignment.

### Domain Assignment

For simplicity, all nodes will be configured at manufacture time to belong to the zero-length domain. Each node has a single domain table entry, specified with the NEURON C compiler directive `#pragma one_domain`.

### Subnet/Node ID Assignment

All the sensor nodes will be configured on one subnet, and all the actuator nodes will be configured on another subnet. This allows the network management tool to assign node IDs within the subnet sequentially by type as the nodes are installed, and also allows up to 127 nodes of each type. When a service pin message is received, the program ID in the message is examined to determine the type of the node, and the 48-bit NEURON ID is compared against the IDs of the known nodes of that type. If this is a new node, the next node ID is assigned, and an *Update Domain* network management message is sent to the node

## Address Assignment

Each of the nodes has only one address table entry that holds a single group address. This is specified with the NEURON C compiler directive `#pragma num_addr_table_entries 1`. When a group is to be created, the network management tool determines an available group ID and assigns member numbers in order of the receipt of the service pin messages. The network management node keeps a table indicating how many members there are in each group. Up to 63 members may belong to a single group since acknowledged service will be used. An attempt to create a group larger than this is ignored. To assign the node to a group, the *Update Addre*ss network management message is sent to the node. If the node is leaving a previous group, the *Update Group Addre*ss network management message is sent to the other members of the old group to update their member counts. If the node is joining a new group, the same message is sent to the other members of the new group for the same reason.

## Network Variable Selector Assignment

All network variables of the same type will have the same selector. This is created by connecting a pair of nodes with the appropriate connections using the LONBUILDER Developer's Workbench, and then duplicating the network image at manufacture time. The service class will be acknowledged and not configurable by the network management tool.

## Data Structures in the Network Management Tool

There are two tables of nodes, one for up to 127 sensor nodes, and one for up to 127 actuator nodes. The tables contain six bytes of 48-bit NEURON ID information, and a single byte indicating which group this node belongs to. There is also a table for up to 255 groups. Each entry consists of a single byte indicating the number of members in this group. The total LON database is therefore 2 x 127 x ( 6 + 1 ) + 255, or approximately 2K bytes of non-volatile memory.

## Implementation

Figure 10 shows a flow chart of the messages received and sent by the network management tool. Each of the messages is sent using a different message tag so that their completion and success events may be handled appropriately. For a detailed description of the use of explicit messaging and explicit addressing of messages, see the *NEURON C Programmer's Guide*. Listing 2 shows the complete code of the NEURON C application program in the network management tool.

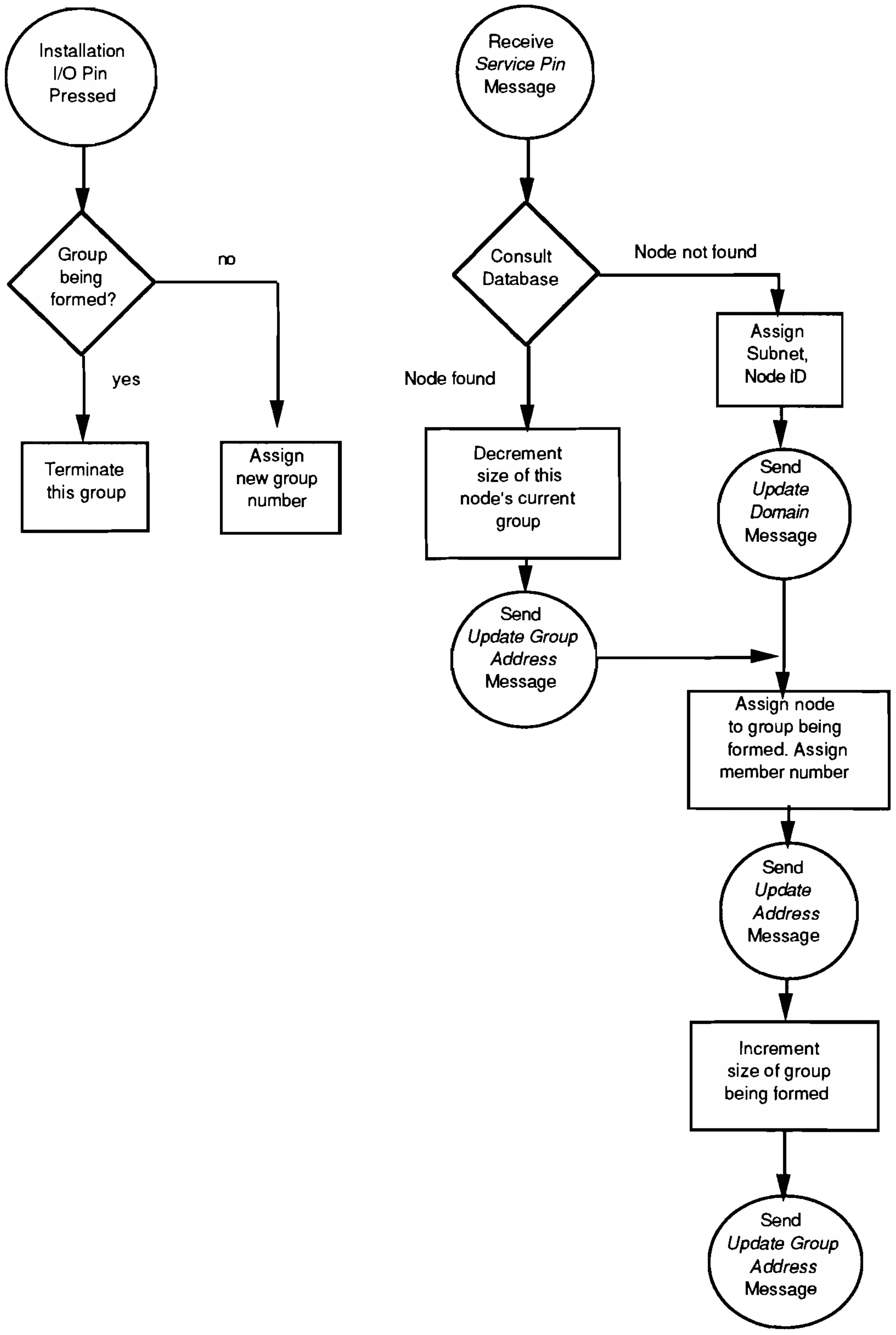

Installation I/O Pin Pressed
Group being formed?
no
yes
Terminate this group
Assign new group number
Receive *Service Pin* Message
Consult Database
Node not found
Node found
Assign Subnet, Node ID
Send *Update Domain* Message
Decrement size of this node's current group
Send *Update Group Address* Message
Assign node to group being formed. Assign member number
Send *Update Address* Message
Increment size of group being formed
Send *Update Group Address* Message

Figure 10. Network management tool flow chart

```
//////////////////// Include Files ///////////////////////////

#include <addrdefs.h>
#include <access.h>
#include <msg_addr.h>
#include <netmgmt.h>

///////////////////// Non-Volatile Database ///////////////////

typedef enum {
    sensor,
    actuator,
    NUM_CLASSES,
} node_type;                // two classes of node in database
                            // nodes are on different subnets
const unsigned SENSOR_SUBNET = 10;
const unsigned ACTUATOR_SUBNET = 11;

typedef struct {
    unsigned    neuron_id[ NEURON_ID_LEN ];
    unsigned    group_id;
} node_rec_type;            // database record for each node

#define NUM_NODES  10 /* Number of nodes of each class - may be up to 127 */
#define NUM_GROUPS 10 /* May be up to 255 */

eeprom far node_rec_type node_rec[ NUM_CLASSES ][ NUM_NODES ];
            // Node ID 0 not used

eeprom far unsigned group_size[ NUM_GROUPS ];  // size of each group
                                    // group ID 0 not used
eeprom far unsigned num_nodes[ NUM_CLASSES ];
                                    // number of members in each class

/////////////////// Input Output Declarations ///////////////

// enable pull-up resistors for push-button
#pragma enable_io_pullups

IO_4  input bit IO_install_button;
IO_0 output bit IO_install_LED = 0;     // initially off

const int ON = 1;       // for LED
const int OFF = 0;
#define PUSHED  1       /* for button */

////////////////// Local Variables //////////////////////////////

node_type       node_class;
unsigned        subnet_id;
unsigned        node_id;
unsigned        group_being_formed = 0;
unsigned        current_group_size;
node_rec_type   node_rec_copy;      // copy of database record
NM_service_pin_msg svc_pin_msg;     // copy of service pin message
boolean         busy_installing = FALSE;
```

```
////////////////////// Define protocol timer values //////////////

// timers for writing domain table entries - 300 msec write time
const unsigned RPT_TIMER_DT = 0;
const unsigned RETRY_DT = 3;
const unsigned RCV_TIMER_DT = 7;     // 1536 msec = 4 * 384 msec
const unsigned TX_TIMER_DT = 9;      // 384 msec

// timers for writing address table entries - 100 msec write time
const unsigned RPT_TIMER_AT = 0;
const unsigned RETRY_AT = 3;
const unsigned RCV_TIMER_AT = 5;     // 768 msec = 4 * 192 msec
const unsigned TX_TIMER_AT = 7;      // 192 msec

// timers for writing RAM network variables - fast write time
const unsigned RPT_TIMER_AP = 0;
const unsigned RETRY_AP = 3;
const unsigned RCV_TIMER_AP = 0;     // 128 msec = 4 * 32 msec
const unsigned TX_TIMER_AP = 2;      // 32 msec

// increase input buffer counts to handle group acknowledgements
#pragma net_buf_in_count 15
#pragma app_buf_in_count 15

// timer to leave installation mode automatically after 20 seconds
stimer install_timer;
const unsigned long INSTALL_TIMEOUT = 20;

/////////////////////////  Message Tags ////////////////////////

msg_tag bind_info( nonbind ) new_node_tag;      // configure new node
msg_tag bind_info( nonbind ) upd_old_gp_tag;    // inform old group
msg_tag bind_info( nonbind ) join_group_tag;    // add node to group
msg_tag bind_info( nonbind ) upd_new_gp_tag;    // inform new group

///////////////// Function Prototypes //////////////////

boolean memcmp( const unsigned *, const unsigned *, unsigned len );
        // return TRUE if exact match

void update_group_members( unsigned group_id ); // update group address
void join_new_node( void );                     // configure new node
void join_group( void );                        // add node to group

///////////////////////// Tasks ////////////////////////////////////

#pragma scheduler_reset
        // for receiving messages and responses

when( msg_arrives( NM_service_pin + NM_opcode_base ) ) {

        // Come here when a service pin message has been received

    unsigned        num_nodes_in_class;
    unsigned        old_group_id;

    if( !group_being_formed ) return;      // ignore if not binding
    if( busy_installing ) return;          // ignore if previous not done yet
    busy_installing = TRUE;                 // lockout any new tries
    install_timer = INSTALL_TIMEOUT;        // reset timeout

    memcpy( &svc_pin_msg, msg_in.data, sizeof( NM_service_pin_msg ) );
            // get local copy of service pin message

    if( memcmp( svc_pin_msg.id_string, ( const unsigned * )"SENSOR\x0\x0",
             ID_STR_LEN ) ) {
            // this service pin message came from a sensor node
        subnet_id = SENSOR_SUBNET;
        node_class = sensor;

    } else if( memcmp( svc_pin_msg.id_string, ( const unsigned * )"ACTUATOR",
             ID_STR_LEN ) ) {
           // this service pin message came from an actuator node
        subnet_id = ACTUATOR_SUBNET;
        node_class = actuator;
    }
    else return;            // bogus service pin message received

    // Find this node in the database
    num_nodes_in_class = num_nodes[ node_class ];

    for( node_id = 1; node_id <= num_nodes_in_class; node_id++ ) {
              // look through database for this class

        node_rec_copy = node_rec[ node_class ][ node_id ];
            // make a copy (can't take address of EEPROM objects)

        if( memcmp( svc_pin_msg.neuron_id, node_rec_copy.neuron_id,
             NEURON_ID_LEN ) ) {     // this Neuron ID matches database

            old_group_id = node_rec_copy.group_id;
                // which group was this node in?
            if( old_group_id == group_being_formed ) {
                busy_installing = FALSE;
                return;
            }   // attempt to add node to group it's already in

            node_rec_copy.group_id = 0;
            node_rec[ node_class ][ node_id ] = node_rec_copy;
                // this node no longer a member

            if( --group_size[ old_group_id ] ) {      // one less member
                msg_out.tag = upd_old_gp_tag;
                update_group_members( old_group_id );
                     // tell all the other members of the group
            }
            else join_group( );  // no members left, add node to new group
            return;              // known node, all done here
        }
    }
    // not a known node, add to the database

    if( num_nodes_in_class >= NUM_NODES - 1 ) return;
                         // can't handle any more
    node_id = num_nodes_in_class + 1;      // allocate next node id
    join_new_node( );                      // configure a new node
```

```
}

//////////////////////////////////////////////////////////////////////

when( msg_completes( upd_old_gp_tag ) ) {

    // Come here when members of the old group have been updated

    if( !group_being_formed ) return;   // out if we're not binding
    join_group( );      // have this node join the new group
}

//////////////////////////////////////////////////////////////////////

when( msg_succeeds( new_node_tag ) ) {

    // Come here when a new node has been given its subnet and node IDs

    if( !group_being_formed ) return;   // out if we're not binding

    // now update database with new node's data

    num_nodes[ node_class ]++;      // increment count in this class

    // make a new record in the database for this node

    memcpy( node_rec_copy.neuron_id, svc_pin_msg.neuron_id, NEURON_ID_LEN );
    node_rec_copy.group_id = 0;     // not yet successfully joined group
    node_rec[ node_class ][ node_id ] = node_rec_copy;

    join_group( );      // have this node join the group being formed
}

/////////////////////////////////////////////////////////////////////

when( msg_succeeds( join_group_tag ) ) {

    // Come here when node has been added to the new group

    if( !group_being_formed ) return;   // out if we're not binding

    // Node has joined new group, update database record

    group_size[ group_being_formed ] = current_group_size;
    node_rec[ node_class ][ node_id ].group_id = group_being_formed;
        // Set group ID in EEPROM

    msg_out.tag = upd_new_gp_tag;
    update_group_members( group_being_formed ); // tell all the members
}

/////////////////////////////////////////////////////////////////////

when( msg_completes( upd_new_gp_tag ) ) {
    busy_installing = FALSE;    // done with this node, ready for a new one
}
```

```
//////////////////////////////////////////////////////////////////

when( msg_fails  ) {         // catch-all tasks
}
when( resp_arrives ) {
}
when( msg_arrives ) {       // anything else that might show up
}

////////////////////////////////////////////////////////////////////

when( io_changes( IO_install_button ) to PUSHED ) {

// Toggle install mode - terminate group, or start new group

    if( group_being_formed ) {          // terminate this group
        io_out( IO_install_LED, OFF );  // turn off LED
        group_being_formed = 0;         // finished with this group
        busy_installing = FALSE;        // reset busy flag
    }
    else {                              // start a new group
        io_out( IO_install_LED, ON );   // turn on LED
        install_timer = INSTALL_TIMEOUT;     // start a timeout

// Now look for an unused group ID for the new group

        for( group_being_formed = 1; group_being_formed <  NUM_GROUPS;
            group_being_formed++ )
            if( !group_size[ group_being_formed ] ) return;
                // found an unoccupied group id

        group_being_formed = 0;     // too many groups, ignore it
        io_out( IO_install_LED, OFF );
    }
}

/////////////////////////////////////////////////////////////////////

when( timer_expires( install_timer ) ) {
    group_being_formed = 0;     // no activity, terminate this group
    io_out( IO_install_LED, OFF );
}

////////////////////////// Functions //////////////////////////////

void join_group( void ) {

    // Make the current node join the current group

    static NM_update_addr_request update_addr_msg;

    current_group_size = group_size[ group_being_formed ];
    if( current_group_size >= 63 ) {    // quit if group is full
        busy_installing = FALSE;
        return;
    }
    // Create an update address table entry network mgmt message

    msg_out.dest_addr.snode.type = SUBNET_NODE;
    msg_out.dest_addr.snode.domain = 0;
    msg_out.dest_addr.snode.node = node_id;
    msg_out.dest_addr.snode.rpt_timer = RPT_TIMER_AT;
    msg_out.dest_addr.snode.retry = RETRY_AT;
    msg_out.dest_addr.snode.tx_timer = TX_TIMER_AT;
    msg_out.dest_addr.snode.subnet = subnet_id;
//  msg_out.priority_on = FALSE;
    msg_out.tag = join_group_tag;
    msg_out.code = NM_update_addr + NM_opcode_base;
//  msg_out.authenticated = FALSE;
    msg_out.service = REQUEST;

    update_addr_msg.addr_index = 0;      // first address table entry
    update_addr_msg.member_or_node = current_group_size;
    update_addr_msg.type = 0x80 | ++current_group_size; // one more in group
    update_addr_msg.domain = 0;
    update_addr_msg.rpt_timer = RPT_TIMER_AP;
    update_addr_msg.retry = RETRY_AP;
    update_addr_msg.rcv_timer = RCV_TIMER_AP;
    update_addr_msg.tx_timer = TX_TIMER_AP;
    update_addr_msg.group_or_subnet = group_being_formed;

    memcpy( msg_out.data, &update_addr_msg,
            sizeof( NM_update_addr_request ) );
    msg_send( );
}

////////////////////////////////////////////////////////////////

void update_group_members( unsigned group_id ) {

    // Update all members of the group with the new member count
    // Caller must assign msg_out.tag field

    static NM_update_group_addr_request update_group_msg;

    // Create an update group address table entry network mgmt message

    msg_out.dest_addr.group.type = 1;
    msg_out.dest_addr.group.size = group_size[ group_id ] + 1;
            // add one for the network manager node
    msg_out.dest_addr.group.domain = 0;
    msg_out.dest_addr.group.member = 0;         // all members
    msg_out.dest_addr.group.rpt_timer = RPT_TIMER_AT;
    msg_out.dest_addr.group.retry = RETRY_AT;
    msg_out.dest_addr.group.rcv_timer = RCV_TIMER_AT;
    msg_out.dest_addr.group.tx_timer = TX_TIMER_AT;
    msg_out.dest_addr.group.group = group_id;
//  msg_out.priority_on = FALSE;
//  msg_out.tag assigned by caller
    msg_out.code = NM_update_group_addr + NM_opcode_base;
//  msg_out.authenticated = FALSE;
    msg_out.service = REQUEST;
    update_group_msg.type = 1;
    update_group_msg.size = group_size[ group_id ];
    // domain and member are not used
    update_group_msg.rpt_timer = RPT_TIMER_AP;
    update_group_msg.retry = RETRY_AP;
    update_group_msg.rcv_timer = RCV_TIMER_AP;
    update_group_msg.tx_timer = TX_TIMER_AP;
    update_group_msg.group = group_id;

    memcpy( msg_out.data, &update_group_msg,
            sizeof( NM_update_group_addr_request ) );
    msg_send( );
}

//////////////////////////////////////////////////////////////////////

void join_new_node( void ) {

    // Assign a subnet/node number to a new node

    static NM_update_domain_request update_domain_msg;

    // Create an update domain network management message

    msg_out.dest_addr.nrnid.type = NEURON_ID;
    msg_out.dest_addr.nrnid.domain = 0;
    msg_out.dest_addr.nrnid.rpt_timer = RPT_TIMER_DT;
    msg_out.dest_addr.nrnid.retry = RETRY_DT;
    msg_out.dest_addr.nrnid.tx_timer = TX_TIMER_DT;
    msg_out.dest_addr.nrnid.subnet = subnet_id;
    memcpy( msg_out.dest_addr.nrnid.nid, svc_pin_msg.neuron_id,
            NEURON_ID_LEN );
//  msg_out.priority_on = FALSE;
    msg_out.tag = new_node_tag;
    msg_out.code = NM_update_domain + NM_opcode_base;
//  msg_out.authenticated = FALSE;
    msg_out.service = REQUEST;

    update_domain_msg.domain_index = 0;      // first domain table entry
    memset( update_domain_msg.id, 0, DOMAIN_ID_LEN );
    update_domain_msg.subnet = subnet_id;
    update_domain_msg.must_be_one = 1;       // this bit must be set
    update_domain_msg.node = node_id;
    update_domain_msg.len = 0;
    memset( update_domain_msg.key, 0xFF, AUTH_KEY_LEN );

    memcpy( msg_out.data, &update_domain_msg,
            sizeof( NM_update_domain_request ) );
    msg_send( );
}
```

```
///////////////////////////////////////////////////////////////////

boolean memcmp( const unsigned * src, const unsigned * dst, unsigned len ) {

        // Returns TRUE if exact match between two byte arrays

    unsigned i;

    for( i = 0; i < len; i++ )
        if( * src ++ != * dst ++ )
            return FALSE;
    return TRUE;
}
```

**Listing 2.** Implementation of NEURON CHIP-based installation example

Part Number 005-0022-01 Rev. B

Echelon Corporation
4015 Miranda Avenue
Palo Alto, CA 94304
Telephone (415) 855-7400
Fax (415) 856-6153

Echelon Europe Ltd
105 Heath Street
London NW3 6SS
England
Telephone (071) 431-1600
Fax (071) 794-0532
International Telephone + 44 71 431-1600
International Fax + 44 71 794-0532

Echelon Japan K.K.
AIOS Gotanda Building #808
10-7, Higashi-Gotanda 1-chome, Shinagawa-ku, Tokyo 141, Japan
Telephone (03) 3440-8638
Fax (03) 3440-8639